Trabajo 3

Samuel Cardenete Rodríguez y Juan José Sierra González

2 de junio de 2017

Introducción:

Para la realización de esta práctica obtendremos el ajuste de modelos lineales basados en dos problemas centrados en dos conjuntos de datos diferentes. En primer lugar trabajaremos con un problema de clasificación, basado en el conjunto de datos "South African Heart Disease", para el reconocimiento de enfermedades cardiovasculares en una población de Sudáfrica; y en segundo lugar con un problema de regresión, basado en el conjunto de datos "Los Angeles Ozone", para predecir los niveles de ozono en Los Angeles.

Comenzaremos primeramente abordando el problema de clasificación:

Clasificación: "South African Heart Disease"

En este caso nos encontramos frente a un problema de clasificación, tal y como hemos visto anteriormente. Se trata de un conjunto de datos que clasifica individuos de una población de Sudáfrica, indicando si padecen o no una enfermedad del corazón en función de los hábitos de vida (consumo de tabaco, obesidad, alcohol...). Como primer paso para abordar el problema, leeremos los datos y los dividiremos seleccionando nuestro conjunto de entrenamiento y de prueba.

Lectura de datos:

Procedemos a la lectura de la base de datos de clasificación 'South African Heart Disease'.

```
datos_housing = as.data.frame( read_csv("datos/housing.data"))
```

```
## Parsed with column specification:
## cols(
##   CRIM = col_double(),
##   ZN = col_double(),
##   INDUS = col_double(),
##   CHAS = col_integer(),
##   NOX = col_double(),
##   RM = col_double(),
##   AGE = col_double(),
##   DIS = col_double(),
##   RAD = col_integer(),
##   TAX = col_double(),
##   PTRATIO = col_double(),
##   B = col_double(),
##   LSTAT = col_double(),
##   MEDV = col_double()
## )
```

```
head(datos_housing)
```

```
##      CRIM ZN INDUS CHAS   NOX    RM  AGE    DIS RAD TAX PTRATIO      B
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
##   LSTAT MEDV
```

```
## 1  4.98 24.0
## 2  9.14 21.6
## 3  4.03 34.7
## 4  2.94 33.4
## 5  5.33 36.2
## 6  5.21 28.7
```

Conjuntos de training y test usados

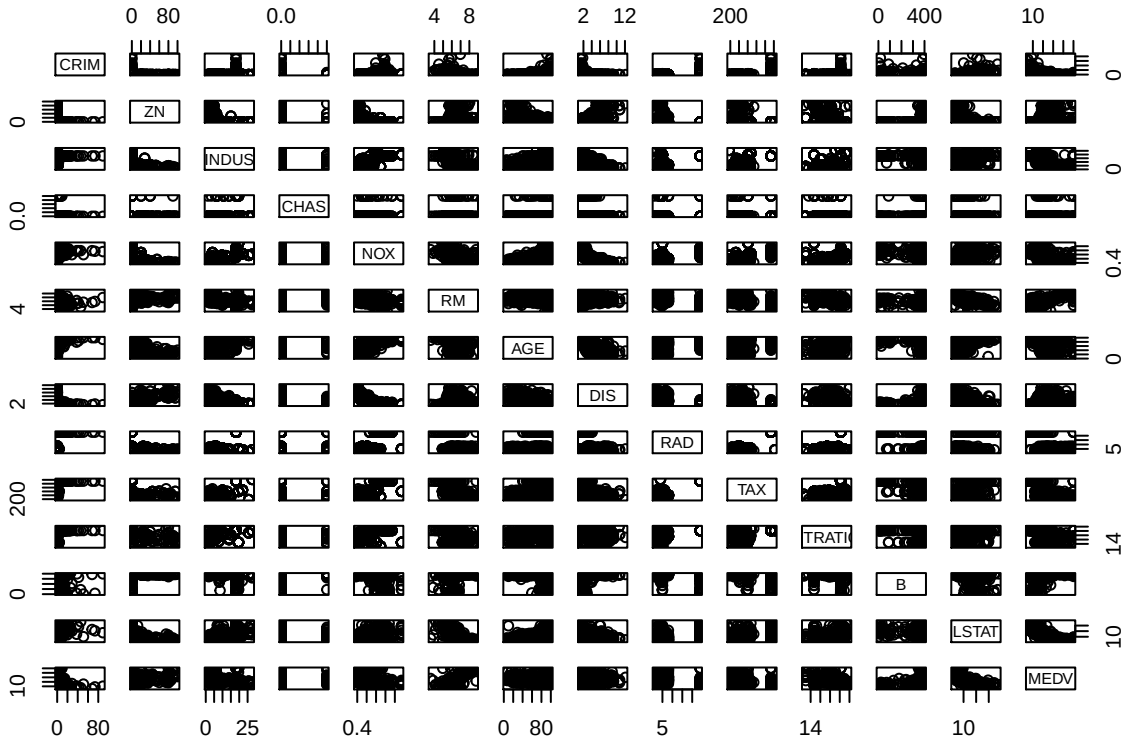
A continuación, realizaremos el particionamiento del conjunto de datos. Para el conjunto 'train' de entrenamiento emplearemos el 80% del conjunto total de los datos, de forma que el 20% restante será empleado para test. Hemos decidido utilizar este porcentaje dado que se nos ha explicado que un buen reparto entre train y test oscila entre dos terceras partes para train, o bien hasta un 80%, utilizando el resto para test. Procedemos al particionamiento de los datos, así como a su almacenamiento en respectivas variables:

```
#Si queremos obtener un conjunto de indices train para luego ejecutar un modelo lineal sobre el train:
train = sample (nrow(datos_housing), round(nrow(datos_housing)*0.8))
#definimos ambos conjuntos en dos data.frame diferentes:
housing_train = datos_housing[train,]
housing_test = datos_housing[-train,]
```

Preprocesamiento de los datos

Antes de entrar en el preprocesamiento, consideramos interesante realizar una vista preliminar de las gráficas de dispersión de cada atributo de la base de datos siendo reflejado con todos los demás. Así obtenemos una matriz simétrica de gráficas, con los colores indicando la etiqueta según padezca enfermedad (rojo) o no (verde):

```
pairs(datos_housing)
```



Al representar la matriz de diagramas podemos hacernos una primera idea de la dispersión de los datos.

Como podemos observar esta dispersión es alta, y a priori los únicos atributos que consideramos que pueden estar sustancialmente relacionados entre sí son 'adiposity' y 'obesity'.

Como estamos tratando un problema de clasificación binaria, generar las gráficas del atributo clase no nos da ninguna ventaja, pero lo hemos dejado en la matriz final para que se pueda apreciar que no aporta ninguna información. Esto es debido a que para todos los atributos, existen casos de ambas clases. De forma contraria, la clase se podría llegar a definir en función de un único parámetro, algo prácticamente inimaginable en un problema real.

Veamos entonces si es posible realizar una reducción de la dimensionalidad de los datos mediante la aplicación de PCA (Computing the Principal Components) sobre el conjunto de datos para intentar comprobar si existen atributos redundantes. Estos atributos redundantes podrían ser recombinados en nuevas características producto de combinaciones lineales de ellos. Prestaremos especial atención a los atributos 'obesity' y 'adiposity' por el motivo indicado anteriormente.

PCA calcula la varianza de cada atributo respecto a los demás, de forma que aquellos atributos que posean menor varianza (cercana a cero) con respecto a los demás serán considerados como redundantes.

Además, para no arriesgar mucho (puesto que PCA trabaja "a ciegas") representaremos sólo aquellos componentes que expliquen hasta el 90% de la variabilidad de los datos (es necesario que los datos estén escalados y centrados para aplicar PCA):

```
housingTrans = preProcess(datos_housing[,-which(names(datos_housing) == "MEDV")], method = c("BoxCox", "
housingTrans$rotation
```

```
##                  PC1          PC2         PC3         PC4          PC5
## CRIM     0.351358922 -0.080323795  0.21229244 -0.03046747  0.033900319
## ZN      -0.255408294  0.026705781  0.40186216 -0.13992202 -0.300421126
## INDUS    0.337559710 -0.035840397 -0.09365446 -0.03972878  0.033764630
## CHAS     0.008802168 -0.567171494 -0.23494015 -0.75574795 -0.160554801
## NOX      0.345753987 -0.204606754 -0.08234027  0.18010421 -0.014959856
## RM      -0.183045703 -0.509627274  0.33596340  0.25887613  0.403709095
## AGE      0.307325638 -0.196513392 -0.22455474  0.24264930  0.001885428
## DIS     -0.326395109  0.246224763  0.13552775 -0.22911923 -0.020886542
## RAD      0.294095534 -0.032632643  0.42351142 -0.20415425  0.178466288
## TAX      0.305780705  0.035027906  0.38197798 -0.16442219  0.059603078
## PTRATIO  0.197381672  0.426164494  0.02183027 -0.33884117  0.523471973
## B       -0.195668297  0.002287144 -0.41362881 -0.10328662  0.556939871
## LSTAT    0.293283963  0.291762870 -0.21021966 -0.04167087 -0.313731923
##                  PC6           PC7
## CRIM    -0.08661682  0.1034686268
## ZN      -0.42440659 -0.5480987095
## INDUS   -0.01982528  0.1706661615
## CHAS     0.12777740 -0.0629487572
## NOX     -0.09608618  0.0004566907
## RM       0.22368487 -0.2592036155
## AGE     -0.03726619 -0.5444595484
## DIS      0.09479850  0.0624644483
## RAD     -0.17887309  0.2017656634
## TAX     -0.29408743  0.0408997952
## PTRATIO  0.39295495 -0.3966467824
## B       -0.66903319 -0.0022056976
## LSTAT   -0.07423922 -0.2975770855
```

Como podemos observar en la tabla, se han reducido los atributos a 8 atributos (combinaciones lineales de los

10 anteriores). Pero si observamos las varianzas de cada atributo en la tabla respecto al resto de atributos, vemos que no existe ningun atributo cuyas varianzas sean cercanas todas a 0. Aún así, para asegurarnos lo comprobamos mediante la siguiente función:

```
nearZeroVar(housingTrans$rotation)
```

```
## integer(0)
```

Como comprobamos con la función nearZeroVar no existe ningun atributo cuyas varianzas respecto a las demás sean todas cercanas a 0, y por tanto todos los atributos son considerados representativos, pues poseen dispersión. En conclusión, no realizaremos ninguna reducción de atributos.

Para conlcuir el preprocesamiento de los datos, realizaremos las siguientes operaciones sobre ellos:

- **Escalado**: Se trata de dividir cada uno de los atributos por su desviación típica.
- **Centrado**: Calculamos la media de cada atributo y se la restamos para cada valor.
- **Box-Cox**: Se trata de intentar reducir el sesgo de cada atributo, para intentar hacer este mas próximo a una distribución Gaussiana.

Para aplicar las transformaciones, emplearemos la función preProcess sobre nuestro conjunto train, de forma que realizando un predict sobre el objeto transformación obtenido, obtengamos el conjunto de datos train preprocesado. A continuación, realizaremos las mismas transformaciones sobre el conjunto de test, utilizando el mismo objeto transformación:

```
housingTrans = preProcess(housing_train[,-which(names(housing_train) == "MEDV")],
                          method = c("BoxCox", "center", "scale"),thresh = 0.9)

housing_train[,-which(names(housing_train) == "MEDV")]=predict(housingTrans,
         housing_train[,-which(names(housing_train) == "MEDV")])

housing_test[,-which(names(housing_test) == "MEDV")] =predict(housingTrans,housing_test[,-which(names(h
```

Estimación de parámetros

Antes de realizar un modelo, veamos cuáles son las características más representativas (las que ofrecen varianza mayor con respecto al resto de datos), de forma que no empecemos a realizar modelos a ciegas, sino fijándonos en la calidad de sus atributos.

Para ello emplearemos la función regsubsets. Esta función realiza una búsqueda exhaustiva (empleando Branch&Bound) de las mejores agrupaciones de atributos en nuestro conjunto de entrenamiento para predecir en una regresión lineal:

```
regsub_housing =regsubsets(datos_housing[,-which(names(datos_housing) == "MEDV")],
                           datos_housing[,which(names(datos_housing) == "MEDV")])

summary(regsub_housing)
```

```
## Subset selection object
## 13 Variables  (and intercept)
##         Forced in Forced out
## CRIM        FALSE      FALSE
## ZN          FALSE      FALSE
## INDUS       FALSE      FALSE
## CHAS        FALSE      FALSE
## NOX         FALSE      FALSE
```

```
## RM          FALSE      FALSE
## AGE         FALSE      FALSE
## DIS         FALSE      FALSE
## RAD         FALSE      FALSE
## TAX         FALSE      FALSE
## PTRATIO     FALSE      FALSE
## B           FALSE      FALSE
## LSTAT       FALSE      FALSE
## 1 subsets of each size up to 8
## Selection Algorithm: exhaustive
##          CRIM ZN  INDUS CHAS NOX RM  AGE DIS RAD TAX PTRATIO B   LSTAT
## 1  ( 1 ) " "  " " " " "   " "  " " " " " " " " " " " "     " " "*"
## 2  ( 1 ) " "  " " " " "   " "  " " "*" " " " " " " " " "     " " "*"
## 3  ( 1 ) " "  " " " " "   " "  " " "*" " " " " " " " " "*"     " " "*"
## 4  ( 1 ) " "  " " " " "   " "  " " "*" " " "*" " " " " " "*"     " " "*"
## 5  ( 1 ) " "  " " " " "   " "  "*" "*" " " "*" " " " " " "*"     " " "*"
## 6  ( 1 ) " "  " " " " "   "*"  "*" "*" " " "*" " " " " " "*"     " " "*"
## 7  ( 1 ) " "  " " " " "   "*"  "*" "*" " " "*" " " " " " "*"     "*" "*"
## 8  ( 1 ) " "  "*" " " "   "*"  "*" "*" " " "*" " " " " " "*"     "*" "*"
```

En la gráfica aportada por la función, obtenemos los atributos más representativos en caso de realizar un modelo con 'n' parámetros (filas), de forma que según el número de atributos que deseamos para nuestro modelo nos estíma cuales serían los parámetros más representativos a emplear. Nuestro criterio a seguir a la hora de generar modelos lineales será, no obstante, tratar de predecir utilizando las agrupaciones de atributos que la función considera representativos, Es decir, si queremos realizar un modelo con n características, utilizaremos aquellas que estén marcadas con una estrella para la fila n. Para mostrar un ejemplo que explique la gráfica, si queremos utilizar 3 atributos para el modelo, escogeremos 'age', 'famhist' y 'tobacco'.

Ahora que sabemos cómo elegir las características más recomendables para realizar modelos, podremos construir una serie de ellos con algunas de estas características y validar con el conjunto de test para comprobar los errores que reflejan.

Regularización

Procedamos ahora a realizar un análisis para ver si es interesante aplicar una regularización empleando Weight-decay mediante la función glmnet. Dicha función recibe los siguientes hiperparámetros:

- **Alpha**: Para aplicar el weight-decay utilizaremos dicho argumento con valor 0.
- **Lambda**: Parámetro de regularización. (Multiplica la matriz de identidad)

Hace falta tener en cuenta antes la correcta elección del lambda, de forma que escojamos el que mejores resultados nos pueda arrojar. En lugar de seleccionarlo de forma arbitraria, será mejor emplear validación cruzada:

```
etiquetas = housing_train[,which(names(housing_train) == "MEDV")]
tr = housing_train[,-which(names(housing_train) == "MEDV")]
tr = as.matrix(tr)
crossvalidation =cv.glmnet(tr,etiquetas,alpha=0)
print(crossvalidation$lambda.min)
```

```
## [1] 0.7884988
```

Una vez obtenido el lambda que proporciona un menor E_{out}, procedemos a generar un modelo de regularización, en primer lugar empleando el valor de lambda generado por validación cruzada, y en segundo lugar empleando un lambda igual a cero, de forma que no apliquemos regularización. El objetivo será comprobar si los

parámetros obtenidos son significativamente diferentes como para que merezca la pena realizar regularización en nuestros modelos.

```
modelo_reg = glmnet(tr,etiquetas,alpha=0,lambda=crossvalidation$lambda.min)
print(modelo_reg)
```

```
##
## Call:  glmnet(x = tr, y = etiquetas, alpha = 0, lambda = crossvalidation$lambda.min)
##
##      Df   %Dev Lambda
## [1,] 13 0.7692 0.7885
```

Aplicando regularización con el lambda obtenido tras la validación cruzada obtenemos una desviación del 0.237. Probemos ahora no aplizando regularización, empleando un hiperparámetro lambda de 0, y comprobemos su desviación:

```
modelo_reg = glmnet(tr,etiquetas,alpha=0,lambda=0)
print(modelo_reg)
```

```
##
## Call:  glmnet(x = tr, y = etiquetas, alpha = 0, lambda = 0)
##
##      Df   %Dev Lambda
## [1,] 13 0.7779      0
```

Como podemos comprobar, las desviaciones obtenidas empleando o no regularización mediante weight-decay son similares (0.237 frente a 0.241), por tanto, comcluimos en que emplear regularización no merece la pena.

Definición de modelos

Para la realización de los modelos planteamos emplear regresión logística mediante la función 'glm'; pero si empleamos regresión logística obtendremos un conjunto de probabilidades que predigan los valores. Para ello, como precondición sería necesario normalizar los valores de la variable respuesta de forma que se encontraran en el intervalo [0,1].

Entonces procederemos a generar modelos mediante regresión lineal, así obtendremos un valor dependiendo de la distancia de los atributos a la recta generada.

```
m_muestra_housing = lm(MEDV ~ LSTAT, data=housing_train)
```

Antes definimos una función para el cálculo del error. Para ello realizamos un cálculo de la media de errores normalizados en un intervalo. Se realiza el cociente entre la resta de las variables respuesta reales menos las predichas, y la diferencia entre el valor máximo y mínimo de las variables respuesta:

```
calculoErrorMedioIntervalo  = function (modelo, test, variable_respuesta){
  prob_test = predict(modelo, test[,-which(names(test) == variable_respuesta)])

  etest = mean(abs(prob_test - test[,which(names(test) == variable_respuesta)])/(max(test$MEDV)-min(tes

}
```

Utilizamos nuestra función para calcular el error del modelo de muestra generado anteriormente:

```
etest_mmuestrasud = calculoErrorMedioIntervalo(m_muestra_housing, housing_test, "MEDV")
etest_mmuestrasud
```

```
## [1] 0.07623536
```

Obtenemos un error de 0.337, pero se trata de un modelo excesivamente simple como para reflejar buenos resultados en un problema real. Por tanto busquemos un modelo diferente empleando otra caraster´ıstica, la siguiente más representativa en el conjunto de datos, que en nuestro caso es 'famhist':

```
m1_housing = lm(MEDV ~ LSTAT + PTRATIO, data=housing_train)

etest_m1sud = calculoErrorMedioIntervalo(m1_housing, housing_test, "MEDV")
etest_m1sud
```

```
## [1] 0.07357477
```

Utilizando dos características el error en el conjunto de test desciende hasta un 0.326. Seguimos probando nuevos modelos, así que añadimos la siguiente característica recomendada por regsubsets, 'tobacco':

```
m2_housing = lm(MEDV ~ LSTAT+ RM+ PTRATIO, data=housing_train)

etest_m2sud = calculoErrorMedioIntervalo(m2_housing, housing_test, "MEDV")
etest_m2sud
```

```
## [1] 0.07302149
```

Un error reflejado de 0.293 ya se acerca más a lo que buscamos, poco a poco vamos avanzando hacia un error menor en el conjunto de validación. Como aún nos queda una buena cantidad de características por probar, añadimos una más al siguiente modelo, 'ldl':

```
m3_housing = lm( MEDV ~ LSTAT+ RM+ PTRATIO + DIS, data=housing_train)

etest_m3sud = calculoErrorMedioIntervalo(m3_housing, housing_test, "MEDV")
etest_m3sud
```

```
## [1] 0.07048815
```

En esta ocasión tenemos un error de 0.261, que mejora sustancialmente al que teníamos antes. Probemos con la siguiente característica, 'typea':

```
m4_housing = lm(MEDV ~ LSTAT+ RM+ PTRATIO + DIS + NOX, data=housing_train)

etest_m4sud = calculoErrorMedioIntervalo(m4_housing, housing_test, "MEDV")
etest_m4sud
```

```
## [1] 0.07078847
```

————AQUI HAY SOBREAJUSTE———— El error se reduce a 0.25, lo podemos empezar a considerar un error aceptable. No obstante, sigamos probando a añadir la siguiente característica según regsubsets, 'obesity':

```
m5_housing = lm( MEDV ~ LSTAT+ I(RM^2)+ PTRATIO + DIS, data=housing_train)

etest_m5sud = calculoErrorMedioIntervalo(m5_housing, housing_test, "MEDV")
etest_m5sud
```

```
## [1] 0.06978351
```

Aquí encontramos el primer bache, y es que aumentando el número de características empeoramos el error en el test. Seguramente sea debido a sobreajuste, pero para asegurarnos vamos a sustituir 'obesity' por el siguiente atributo más válido según regsubsets, 'sbp':

```
m6_housing = lm( MEDV ~ LSTAT* RM* PTRATIO * DIS, data=housing_train)

etest_m6sud = calculoErrorMedioIntervalo(m6_housing, housing_test, "MEDV")
etest_m6sud
```

```
## [1] 0.05175621
```

Efectivamente, seguimos encontrando errores peores, por lo que abandonamos esta línea de exploración al estar enfrentándonos a una base de datos no lineal. Para mejorar el error hemos realizado diferentes transformaciones no lineales sobre los atributos seleccionados como más representativos.

MODELOS NO LINEALES:

Comenzamos a aplicar modelos:

REDES NEURONALES:

El error no es malo pero sigue siendo un peor modelo que el mejor que hemos generado ahora mismo. Hemos comprobado también que es contraproducente realizar transformaciones con logaritmos y raíces debido a la cantidad de datos negativos que tiene la base de datos una vez normalizada.

En resumen, tras realizar distintas combinaciones de atributos y tratar de predecir con ellos, utilizando alguna transformación no lineal, experimentalmente hemos reducido el error fuera de la muestra a un 0.239. Este error se ha obtenido con un modelo con transformación cuadrática sobre el atributo 'age' (el que mejor representa la muestra como hemos visto en regSubsets) y utilizando los 4 siguientes mejores atributos.

Estimacion del error E_{out}

Una vez que hemos comprobado qué modelo nos aporta un mejor resultado, vamos a proceder a realizar una estimación del E_{out} de forma que obtengamos un valor más representativo que un error puntual en un par de conjuntos train-test.

Para ello vamos a realizar un experimento repitiendo el proceso completo, desde la generación de particiones, tanto train como test, con la misma proporción realizada anteriormente (0.8 para train y 0.2 para test), hasta la definición de los modelos y cálculo del E_{test}. Repetiremos el proceso 100 veces y obtendremos la media de los E_{test}, considerándolo una buena estimación del E_{out} para el modelo.

Definamos una función que realice tanto la generación y el particionado, así como las transformaciones sobre los conjuntos de datos y que devuelva el error generado por el mejor modelo obtenido:

```
generarErrorParticionhousing = function(datos){
  #Si queremos obtener un conjunto de indices train para luego ejecutar un modelo lineal sobre el train
  indices_train = sample (nrow(datos), round(nrow(datos)*0.7))
  #definimos ambos conjuntos en dos data.frame diferentes:
  housing_train = datos[indices_train,]
  housing_test = datos[-indices_train,]

  #TRANSFORMACIONES:
  housingTrans = preProcess(housing_train[,-which(names(datos_housing) == "MEDV")], method = c("BoxCox"
housing_train[,-which(names(housing_train) == "MEDV")] =predict(housingTrans,housing_train[,-which(name
housing_test[,-which(names(housing_test) == "MEDV")] =predict(housingTrans,housing_test[,-which(names(h

#EVALUACION DEL MODELO
modelo_housing = lm(MEDV ~ LSTAT* RM* PTRATIO * DIS, data=housing_train)

etest = calculoErrorMedioIntervalo(modelo_housing, housing_test, "MEDV")
etest
}
```

```r
mean(replicate(100, generarErrorParticionhousing(datos_housing)))
```

```
## [1] 0.0590335
```

Como podemos ver, obtenemos una estimación del E_{out} de 0.286. Este error representa mejor el error real obtenido por nuestro modelo de regresión lineal, aunque sea peor que el error puntual que teníamos antes. Seguramente esto se deba a que para los conjuntos train y test definidos anteriormente el modelo se ajustaba particularmente bien, mientras que de forma general en el problema ese modelo no llega a ser tan bueno.

Conclusión y modelo final seleccionado